TÉCNICO LISBOA

Highly Dependable Systems

MEIC-A

HDS Ledger - 2nd Stage

Authors:

David Belchior (95550) davidbelchior@tecnico.ulisboa.pt
Diogo Santos (95562) diogosilvasantos@tecnico.ulisboa.pt
Vasco Correia (94188) vasco.cardoso.correia@tecnico.ulisboa.pt

Group 4

2022/2023 – 2nd Semester, P3

Contents

1 Objective

This stage of the project aims to pick up on the first stage's Istanbul BFT consensus protocol [3]. Its reliability allows us to create a full-fledged blockchain system, including accounts and transactions between them. These accounts can have their balance checked, both via weakly and strongly consistent read operations.

2 Blockchain and mempool

Like a typical blockchain, each received operation is aggregated into a mempool queue. In the case of the leader, each time the its size exceeds the (fixed) block size, a new block is created with the first messages in it and the consensus protocol is started.

2.1 Pre-prepare and prepare stages

The leader sends a pre-prepare stage including the newly-formed block to all nodes. After that, each one replies with a prepare message, piggybacking an ACK to stop the pre-prepare's transmission to that node.

2.2 Block validation and commit stage

Once a node receives a Byzantine quorum of $2f + 1$ prepare messages (f is the maximum number of faulty processes), it checks all the block's write transactions (the read operations aren't relevant) by assessing whether they can be executed over the ledger's current state (more details on that assessment in 3). An invalid operation invalidates the whole block, but to reduce the frequency of invalidations resulting from the unordered reception of operations, all account creations are executed first.

We understand that this solution is vulnerable to Byzantine clients flooding the system with invalid requests, affecting liveness and inducing content censorship (especially when purposefully conflicting with specific clients' requests). We address this in 5.

In any case, the node sends a commit message to $2f+1$ nodes with a map of updates, where the key is the signature and the value is the final (un)changed state of the account.

To use this message as an ACK to received prepare messages, we need to know which message ID we are replying to. As such, we do not broadcast the commit message immediately. Instead, we send it first to the nodes belonging to the received quorum only, and then, for each of the subsequent prepare messages, we reply with the previously committed content.

2.3 Commit stage conclusion

Just like in the prepare stage, the node waits for a quorum of commit messages to decide on a block. Once that happens, the operations are effectively applied, and, along with this consensus instance's account updates, the previously forgotten read operations' results are sent. For that goal, after the transactions are applied, the node looks for each read account's most recent update, and adds it to the final list of block related operations.

Finally, we aggregate these operations by the requesting client and send just one reply with the final account state, signed by the quorum of received signatures for it, saving the overhead related with replying to every single transaction. These signatures will allow the client to verify the correctness of a response after receiving a single reply from the nodes. Note that this message doesn't need to send the block because we are sure that it was already correctly propagated.

We discuss an improvement to the number of nodes that must reply to the client in 5.

2.4 Round change

Although a round change algorithm was not required, we still added a timer per request to inform when a request was kept in the mempool for longer than a specified time limit, which could lead, for example, to a round change in the consensus module.

3 The ledger

This stage's core is the ledger, which aggregates accounts accessible via a hash of the account owner's public key (available in the PKI).

3.1 Temporary and definitive accounts

In order to allow a correct execution of the operations without damaging the safe (i.e., committed) state of the ledger, there is a duplicate version of each account, which we call a temporary account.

All write operations requested by clients are initially applied on the temporary accounts, and only copied to the definitive accounts once the block is committed. Read operations, on the other hand, are directly executed on definitive accounts.

3.2 Write operations

For all write operations, the client sends the request to a Byzantine quorum $(2f + 1)$ of nodes. Moreover, since each reply itself contains a Byzantine quorum of signatures, only one correct reply is required, meaning that, in the worst case, we need a total of $f + 1$ replies. In each of the following operations, a fixed fee (1), located, for simplicity's sake, in the ledger, is paid to the leader.

3.2.1 Account creation

In this operation, the client sends its public key (implicitly added in the request by our library, providing them a useful abstraction layer) and tries to create their account. The operation succeeds if the account didn't previously exist, adding a fixed amount of cash (100) to it, and fails otherwise.

3.2.2 Transfer between accounts

In this operation, the client sends its own (as the sender) and the receiver's public key (implicitly added in the request by our library, providing them a useful abstraction layer) and tries to send the provided amount to the receiving account. The operation succeeds if both accounts exist and the sender's account's balance is higher or equal to the amount to transfer, plus the leader's fee, and fails otherwise.

3.3 Read operations

3.3.1 Genesis block

One of the issues we faced regarding read operations was the lack of signatures when an account didn't exist. For that, before any block containing client requests is sent to the consensus module, i.e, in the initialisation process, one block, the so-called Genesis block [1], containing a "pseudo-creation" of all clients' accounts and the leader's account, is acted upon. This step creates those accounts, albeit in an "inactive" state. Thus, once a client tries to "create" their account, this results in a simple activation.

From the client's perspective, though, "inactive" accounts are translated into "non-existing" accounts, and the opposite for "active" accounts.

3.3.2 Weakly consistent read

In a weakly consistent read, the client contacts $f + 1$ replicas and waits for the first correct reply. In that case, if the replied state is more recent than the one the client already knows (which is kept in the library), the known state is updated and returned; otherwise, the already known state is returned, and the received reply is discarded. As stated before, the client receives $2f + 1$ signatures of the replied account state which he can verify.

3.3.3 Strongly consistent read

Strongly consistent reads work in a maximum of two stages, based on PBFT [2]. In the first stage, the client broadcasts its request, waits for a Byzantine quorum of replies and checks if all those replies are identical. If it's successful, the known state is updated and returned; otherwise, a new read request is issued to a Byzantine quorum, this time directly to the consensus module.

This solution is, on average, quicker than a full instance of consensus. This happens because, in the best case scenario, the client returns after the first stage, with no need for a costly consensus-dependent operation.

4 Testing

To test the system we have multiple node and client configuration files where both can have Byzantine behaviour, such as:

- **Greedy client:** Swap the transfer's source and destination (to itself).
- **Fake weak read**: Send messages with a forged balance as replies to weak reads.
- **Dictator leader:** Ignore a particular client's requests.
- **Silent leader:** Send no messages to start a consensus instance.
- **Landlord leader:** Charge more than the correct fee to each client.
- **Handsy leader:** Modify the mempool's content.
- **Corrupt leader:** Forge and inject transactions that increase its balance into the mempool.
- **Fake leader**: Send messages pretending to be the leader - Since other nodes' signature checks will fail, they will not reply with ACK meaning that this node will be stuck sending messages forever.
- **Force consensus read:** All nodes return incorrect responses on strong reads in order to force consensus.
- **Drop packets**: All received packets are dropped.
- **Bad consensus**: Attempt to start consensus, to test if other nodes only accept messages from the actual leader.
- **Bad broadcast**: Send messages with random values to different nodes.

5 Future improvements

Some improvements were already mentioned during the description of the system, but, to summarise them:

- **Firewall:** Prevent byzantine clients from flooding the mempool with invalid requests
- **Responding nodes:** Only $f + 1$ nodes should respond to the client, since a single response contains the signatures of $2f + 1$ nodes. This could be implemented by having a mapping of consensus instance to node IDs, meaning that for each instance a different set of nodes would reply to the client.

References

[1] Definition of the Genesis block in Bitcoin. https://www.blockchain.com/explorer/blocks/btc/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f. Accessed: 2023/04/13.

[2] Miguel Castro, Barbara Liskov, et al. Practical Byzantine Fault Tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[3] Henrique Moniz. The Istanbul BFT consensus algorithm. *arXiv preprint arXiv:2002.03613*, 2020.